ATARI Disk Data Structures: An Interactive Tutorial

by David Young

INTRODUCTION

The floppy disk is a marvelous and yet mysterious media for mass storage of data. Indeed, to understand exactly how a bit of data is stored and retrieved from the surface of the disk requires a good knowledge of physics. However, to learn about the data structures found on a disk requires no higher mathematics than hexadecimal arithmetic. The manual supplied with the computer usually does an adequate job of supplying all the technical details, but wouldn't it sink in better if the actual data on the media were viewed while it is being described? The DISKSCAN program is used to demonstrate the disk data structures as they are being described. Follow the instructions under GETTING STARTED in the DISKSCAN USER'S GUIDE to run DISKSCAN. Once the program has started, remove the DISKSCAN diskette and insert some other diskette that has been backed up. Use the "R" (read sector) function whenever you are requested to view a particular sector on disk. Whenever you are requested to change the display format from hex to character, or vice versa, use the "T" (toggle display format) function.

The Disk Media

The first disk structure to be aware of is the sector, which on any computer system consists of a group of contiguous bits recorded at a specific location on the disk. The disk drive hardware always operates on whole sectors, that is to say, it is not possible to read or write partial sectors. Groups of sectors are organized into tracks forming concentric rings about the center of the disk. The ATARI system divides the disk into 40 tracks with 18 sectors per track for a total of 720 sectors. This is best visualized by taking the lid off of the disk drive and watching the read/write head move as certain sectors are addressed. On the ATARI 810 disk drive this is accomplished by removing the 4 phillips head screws hidden under gummed tabs at each corner of the lid. While inside the case, a bit of lubrication on the 2 cylindrical guide rails supporting the head will make the drive less noisy.

If sectors 1 through 18 are read with DISKSCAN, the head remains fixed on the outermost track. When sector 720 is read, the head moves in to the innermost track. When a disk is formatted, the head can be seen to bump sequentially through all 40 tracks. It is laying down the patterns on the oxide surface which will be recognized by the drive hardware as the sectors. The sectors are all initially empty (128 bytes of 0), but at the end of the formatting routine, as described in the next section, the ATARI DOS records special data into certain sectors. The top of the drive can now be resecured. No more information about the hardware is needed to understand the higher level disk data structures of the software.

Boot Sector

At the end of the formatting process DOS reserves and initializes certain sectors for special tasks. Into sectors 1 through 3 is stored the bootstrap for DOS. On power-up the ATARI operating system reads sector 1 to determine how many sectors to read and where into memory to load them. After it has loaded in the specified number of sectors, DOS starts executing the new code at the load address + 6. Put DISKSCAN into the hex mode and read sector 1 of any DOS disk. Byte 1 says that 3 sectors are read (sequentially) and bytes 2 and 3 specify a load address of $700. (A 2 byte number is always specified with the least significant byte first.) Byte 6 is the first intruction to be executed (a $4C1407 is a JMP $714). In this case the code which follows sets up to load the File Management System of DOS into memory. This is called the second stage of the boot. Look at the first sector of any other boot disk available (any game or program which loads in from disk on power-up). It might be seen that the program loads in entirely during the first stage of the boot, i.e. byte 1 of sector 1 has a sector count which represents the entire program. For more details on the disk boot process, see the ATARI Operating System User's Manual.

Volume Table of Contents

Besides the first three boot sectors, DOS sets up sectors 360 to 368 as the directory of the disk. DOS uses the directory to keep track of where files are stored on disk and how much disk space remains. Read sector 360 of a DOS disk with DISKSCAN in the hex mode and view a part of the directory called the Volume Table of Contents (VTOC).

Information pertaining to the availability of every sector on the disk is stored in this sector. Bytes 1 and 2 specify the maximum number of user data sectors on the disk ($2C3 = 707) and bytes 3 and 4 specify the number of free sectors remaining on the disk (707 for an empty disk, 0 for a full one). Starting in bit 6 (the second to highest order bit) of byte $0A, each bit up through byte $63 corresponds to a sector. A 1 corresponds to a free sector while a 0 means the sector is being used.

When a file is stored on the disk, the bits corresponding to the sectors used are set to 0. When the file is erased, the bits are set back to 1. That is why DOS, when it deletes a file, can be heard reading the entire file. It is determining which sectors were being used by the file so that it can free them back up. Notice that bits 1, 2 and 3 (bits 6, 5 and 4 of byte $0A) are set to 0. These correspond to the 3 boot sectors. Likewise, the 9 bits starting in byte $37 are 0 because they correspond to the sectors of the directory. These 12 sectors are thus kept from being overlaid by user files.

If the VTOC is viewed on an older disk which has had many file additions and deletions, it may be noted that the VTOC has become quite fragmented. Any file added to the disk may get stored into sectors scattered about the disk. How DOS keeps track of files spread over multiple sectors will be discussed shortly. By the way, even though the operating system recognizes sector 720 (try reading it; should be all zeroes), DOS never makes use of it. True to Murphy's Law, it adopted the number scheme of 0 to 719 instead of 1 to 720. No need to bother trying to read sector 0!

The Directory

Of all the disk data structures, probably the most important one to be acquainted with is the directory. The 8 sectors following the VTOC (361-368) contain a list of all the files on the disk along with their size, starting sector and status. Put DISKSCAN into character mode and read sector 361 of a

DOS disk that has several files on it. It can be seen that the name of the first file starts in byte $05 and the extension (if any) starts in byte $0D. If any of the 11 character positions of the filespec are unused, it contains a blank. Notice that the filenames start every 16 bytes, allowing 8 directory entries per 128 byte sector. Thus, the maximum number of entries for the 8 sectors of the directory is 64.

Now put DISKSCAN in hex mode and read sector 361. The first byte of each 16 byte entry contains the status of the file. For a normal file that byte is $42, unless it is locked, in which case it has a status of $62. A deleted file has a status of $80. An anomaly occurs whenever a file is opened for output (from BASIC, perhaps) but is not closed before the computer is powered down or glitched. Since the status of an open file is $43, DOS will neither recognize the entry as "in use" nor "deleted". Even the sectors which may have been written out will not really exist on disk because the VTOC is not updated until the file is closed. The only harm done is that this bogus entry will take up space in the directory until the disk is reformatted. (One other solution would be to change the $43 to an $80 using DISKSCAN; refer to the change sector function, "C", in the DISKSCAN USER'S GUIDE.) The second and third bytes of each entry contain the size in sectors of the file (low order byte first) while the fourth and fifth bytes specify the first sector of the file. DOS only needs to know the first sector of a file because each sector points to the next sector of the file in a process called "linking".

Linking

At this point it would be best to explain how DOS forms a data file on disk. First, the user must open an I/O channel for output to the disk, perhaps with the BASIC "OPEN" command. DOS responds by creating an entry in the directory with the specified filename and a status of $43. DOS reads the VTOC into memory and searches the disk map for the first free sector. If a free sector is found, it's number is used as the starting sector in the directory entry. Now, when the user begins to output data via this I/O channel, perhaps with the BASIC "PUT" command, DOS waits until it has collected 125 bytes of user data in a buffer. Then DOS adds 3 special bytes of it's own and outputs the sector to the disk. I call these 3 bytes the "sector link".

The sector link, bytes 125 to 127 of the sector, contains 3 pieces of information. The high order 6 bits of byte 125 contain a number which represents the position of the file's entry within the directory (0 to 63). DOS uses this number to check the integrity of the file. If ever this number should fail to match the position of the file's directory entry, DOS generates an error. The low order two bits of byte 125 and all of byte 126 form a pointer to the next sector of the file. A pointer is the address of a record in the computer's memory or, in this case, the address of a record on disk, the sector number. The next sector of the file is determined by scanning the bit map of the VTOC for the next free sector, which may or may not be the next sequential sector of the disk. Thanks to the link pointers, [illegible] sectors of a file need not be contiguous sectors on the disk. The last byte of the sector link (byte 127 of the sector) contains the number of bytes used within the sector. This byte will always be $7D (125) except for the last sector of a file, which will probably be only partially filled. DOS writes out this partial sector only when the user closes the file, perhaps with the BASIC "CLOSE" command.

When an output disk file is closed, DOS writes the newly updated VTOC back out to sector 360. It then updates the file's directory entry by changing the status to $42 and filling in the file size (bytes 1 and 2) with the number of sectors used by the file. This completes the process of creating a file on disk. Now, when DOS is requested to read a file from disk, it finds the directory entry of the specified file to determine the start sector. Then, following the link pointers, it reads the file sector by sector until EOF (end of file) is reached, indicated by a link pointer of 0.

Equipped with a basic understanding of how a file is stored on disk, try looking at a file with DISKSCAN. In character mode, first locate the name of the desired file in the directory (sectors 361-368). Then put DISKSCAN in hex mode and look at the fourth and fifth byte of the entry to determine the start sector. For example, if these two bytes were "1F 01", type "$10F" in response to "Sector #?" to read the first sector of the file. Observe the last three bytes of the sector and verify that the high order 6 bits of byte 125 correspond to the directory entry position and that byte 127 is the number of bytes used (probably $7D). Then determine the next sector of the file from the low order 2 bits of byte 125 and byte 126. For example, if bytes 125 and 126 are "1D 20" then the next sector of the file is $120 and the file is the eighth entry of the directory (the first entry being entry 0). If the file is not too long, it would be instructive to follow the sector links to EOF. Once the ability of finding a file on disk and following the sector links is mastered, all that remains is to become familiar with the 3 types of files used by DOS. (NOTE: DISKSCAN will automatically follow the sector links of a file if it is in linked mode; refer to the DISKSCAN USER'S GUIDE.)

File Types

The first type of file is not a true file, per se, because there is no entry in the directory for it. This file type includes the boot record and the directory itself. And since the sectors which make up these files are not linked but, instead, are related to each other sequentially, I call these records "sequentially linked files". When examining a sector of the boot record or directory, merely increase the sector number by 1 to get to the next sector of the record.

An example of the second type of file is that which is created with the BASIC "LIST" or "SAVE" command. This file consists of ASCII characters which either represent straight text, as in a LISTed file, or a sort of condensed text, as in a tokenized or SAVEd file. Except when viewing the sector links, the character mode of DISKSCAN is best suited for examinimg this type of file. At this point it would be instructive to locate in the directory of a DOS disk a file created with the BASIC "LIST" command. Upon determining the start sector, observe the file in the character mode. The BASIC program can be easily recognized. It may be noted that the carriage return-line feed character (CRLF) is displayed in it's ATASCII representation (an inverse escape character) instead of executed. Now observe a file that consists of a program that was SAVEd from BASIC. Since the text has been tokenized the program is harder to recognize. However, certain parts of the program are not altered during the tokenization process, notably text following REM and PRINT statements. Now, having investigated ASCII files, it is time to discuss the last file type, the "binary load" file.

The binary load file is primarily used to load 6502 machine code into memory for execution. However, it's format is so general that it can be used just as easily to load any type of data, including ASCII text. Locate a game or other program which is run with the BINARY LOAD option of DOS. Alternatively, create a binary load file by saving any part of memory (except ROM) with the BINARY SAVE option. Now observe the first sector of the file with DISKSCAN in the hex mode. First, notice that all binary load files start with 2 bytes of $FF. The next four bytes are the start and end addresses, respectively, where the data to follow will be loaded into memory. If these four bytes were "00 A0 FF BF" then the data would be loaded between the addresses of $A000 and $BFFF. I call these four bytes a load vector. After DOS has loaded in enough bytes to satisfy the load vector, it assumes, unless EOF is reached, that the next four bytes specify another load vector. DOS will continue inputting the file at this new address.

Upon completion of a BINARY LOAD, control will normally be passed back to the DOS menu. However, DOS can be forced to pass control to any address in memory by storing that 2 byte address at location $2E0. To store the 2 bytes, it is necessary to specify another load vector as part of the file. If, for example, it were desired to execute the program loaded in at $A000, the following load vector would be part of the file: E0 02 E1 02 00 A0. I call this specialized load vector an autorun vector. It achieves the same result as the RUN AT ADDRESS option of DOS. Try to find the autorun vector in the file being viewed. Although it could be at the beginning, it is most [illegible]kely located at the very end of the file.

Conclusion

Anyone who has made it to the end of this tutorial and has successfully performed the exercises presented here should consider themselves proficient on the subject of ATARI disk data structures. I hope this tutorial has been useful to those wishing to gain a perspective about how data is stored on disk. At the very least it should have taken some of the mystery out of working with this popular device.